References and Local State

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

The lack of mutable variables gives us the ability to perform many analyses using mathematics. In this lecture we talk about *equational reasoning* and references, and see techniques for limiting the scope of the state to improve the reliability of your code.

- Be able to explain equational reasoning and give an example.
- Know the syntax of references in OCaml.
- Know the tradeoffs between imperative and functional features.
- Know the constructions to define a function with local state.
- Be able to state the benefits of local state and give an example.
- Be able to use tuples to allow multiple functions access to the same state.

The rule of *referential transparency:*

$$\frac{e_1 \Downarrow v \quad e_2 \Downarrow v \quad f\ e_1 \Downarrow w}{f\ e_2 \Downarrow w}$$

- If you have two expressions that evaluate to be the same thing then you can use one for the other without changing the meaning of the whole program.
- e.g. `f(x) + f(x) == 2 * f(x)`
- You can prove this by induction, using the natural semantic rules from the previous lectures.

- You can use equational reasoning to make the following equivalence:

$$\texttt{f}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \equiv \texttt{if}\ e_1\ \texttt{then}\ \texttt{f}(e_2)\ \texttt{else}\ \texttt{f}(e_3)$$

```
1 x * (if foo then 20 / x else 23 / x) equivalent to
2 if foo then 20 else 23   (well, mostly)
```

- You have the basis now of many compiler optimization opportunities!

A Complication


```
# let counter = something
val counter : unit -> int = <fun>
# counter ();;
- : int = 1
# counter ();;
- : int = 2
# counter ();;
- : int = 3
#
```

- Can we still use equational reasoning to talk about programs now?

A Counterexample

- `f(x) + f(x) == 2 * f(x)`

```
# 2 * counter ();;
- : int = 8
# counter () + counter ();;
- : int = 11
```

- Congratulations. You just broke mathematics.

Transition Semantics

`ref` $v \rightarrow \$i$, where $\$i$ is a free location in the state, initialized to v.

! $\$i \rightarrow v$, if state location $\$i$ contains v

$\$i := v \rightarrow ()$, and state location $\$i$ is assigned v.

$(); e \rightarrow e$

Note that references are different than pointers: once created, they cannot be moved, only assigned to and read from.

$\dfrac{e \Downarrow v}{\texttt{ref}\ e \ \Downarrow\ \$i}$, where $\$i$ is a free location in the state, initialized to v.

$\dfrac{e \Downarrow \$i}{!e \Downarrow v}$, if state location $\$i$ contains v.

$\dfrac{e_1 \Downarrow \$i \quad e_2 \Downarrow v}{e_1 := e_2 \Downarrow ()}$, and location $\$i$ is set to v.

$$\frac{e_1 \Downarrow () \quad e_2 \Downarrow v}{e_1; e_2 \Downarrow v}$$

Counter, Method 1

```
# let ct = ref 0;;
val ct : int ref = {contents=0}
# let counter () =
    ct := !ct + 1;
    !ct;;
val counter : unit -> int = <fun>
# counter ();;
- : int = 1
# counter ();;
- : int = 2
```

Bad Things for Counter

`ct` is globally defined. Two bad things could occur because of this.

1. What if you already had a global variable `ct` defined?
   - Correct solution: use modules.
2. The Stupid User^{TM} might decide to change `ct` just for fun.
   - Now your counter won't work like it's supposed to...
   - Now you can't change the representation without getting tech support calls.
   - Remember the idea of *abstraction*.

Conclusions about State

State is bad because:

- it breaks our ability to use equational reasoning
- users can get to our global variables and change them without permission

State is good because:

- Certain constructs are almost impossible withouth state (e.g., Graphs)
- Our world is a stateful one

Review of scope:

```
1 let x = 10;;
2
3 let foo y = match y with
4  | 0,b -> let c = b * b in
5           let d = c * c in
6              b * c * d
7  | a,b -> map (fun z -> z + a + x) [a;b]
```

- x exists from line 2–7.
- y exists from line 3–7.
- b exists from line 4–6.
- c exists from line 5–6.
- d exists on line 6 only.
- a and b exist on line 7 only.
- z exists on line 7, after the `fun z` until the).

Using local state

```
# let counter =
    let ct = ref 0 in
    fun () -> ct := !ct + 1; !ct;;
val counter : unit -> int = <fun>
# counter ();;
- : int = 1
# counter ();;
- : int = 2
```

- This protects `ct`, making it available only to `counter`.

Random Number Generators


```
# let mkRandom s =
      fun () -> s := (!s * 9 + 5) mod 1024; !s;;
 val mkRandom : int ref -> unit -> int = <fun>
# let rnd0 = mkRandom (ref 1);;
val rnd0 : unit -> int = <fun>
# rnd0 ();;
- : int = 14
# rnd0 ();;
- : int = 131
# rnd0 ();;
- : int = 160
```

- In this version we pass the reference into the function rather than creating our own.

Function Tuples

```
# let (counter, reset) =
    let ct = ref 0 in
      (fun () -> ct := !ct + 1; !ct),
      (fun nv -> ct := nv);;
val counter : unit -> int = <fun>
val reset : int -> unit = <fun>
# counter ();;
- : int = 1
# reset 5;;    (* This trick brought to you by *)
- : unit = ()  (* higher order functions, tuples, *)
# counter ();; (* and the principle of orthogonality. *)
- : int = 6
```

```
# let enumerate lst (ctfun, rsfun) =
    rsfun 0;
    List.map (fun x -> (ctfun (), x)) lst;;
val enumerate : 'a list ->
  (unit -> 'b) * (int -> 'c) -> ('b * 'a) list = <f
# enumerate ["hello";"there";"class"]
            (counter, reset);;
- : (int * string) list = [1, "hello"; 2, "there";
                           3, "class"]
#
```

- We can give the counter to another function.
- This is not good. Why not?

1. Supposing you wanted a counter that did *not* use references, how would you go about writing it?
2. The random number function generator does not have a way to reset the state. We would also like to be able to ask “what was the last random number generated” without changing the seed. Write a (group of) functions to do this.
3. Suppose we want a more generic way to represent counters—in fact, suppose you want *several* counters in your program. You could just repeat the code several times, but there are serious flaws to that approach. What are they? How might you go about fixing them?

Answer to 2


```
# let mkRandom init =
    let seed = ref init in
  (( fun () -> seed := !seed * 4; !seed),
    ( fun ns -> seed := ns),
    ( fun () -> !seed) );;
        val mkRandom : int -> (unit -> int) * (int
# let (rnd, reset, last) = mkRandom 4;;
val rnd : unit -> int = <fun>
val reset : int -> unit = <fun>
val last : unit -> int = <fun>
```